Technische Universität Berlin

Forschungsberichte
der Fakultät IV – Elektrotechnik und Informatik

Gruppen: Ein Ansatz zur Vereinheitlichung von Namensbindung und Modularisierung in strikten funktionalen Programmiersprachen (Langfassung)

Florian Lorenzen und Judith Rohloff

Bericht-Nr. 2011 – 12
ISSN 1436-9915

Gruppen: Ein Ansatz zur Vereinheitlichung von Namensbindung und Modularisierung in strikten funktionalen Programmiersprachen (Langfassung)

Florian Lorenzen und Judith Rohloff

Technische Universität Berlin
{florian.lorenzen,judith.rohloff}@tu-berlin.de

No. 2011–12

ISSN 1436-9915

November 28, 2011

Zusammenfassung Das in [1] vorgeschlagene Konzept der Gruppen vereinigt dynamische Datenstrukturen sowie Modularisierungs- und Bindungsaspekte in funktionalen Programmiersprachen. Sie erfassen rekursive Definitionen, lexikalische Sichtbarkeit, hierarchische Programmstrukturierung sowie dynamisch getypte Datenstrukturen in einer gemeinsamen Konstruktion. Wir illustrieren diese Konstruktion an charakteristischen Beispielen und entwickeln eine Analyse und Transformation, die es erlaubt, Gruppen mit einer Call-by-value-Strategie auszuwerten sowie Ihnen eine präzise Semantik zuzuordnen. Zur Ermittlung der Abhängigkeitsreihenfolge erzeugen wir einen spezifischen Abhängigkeitsgraphen, dessen starke Zusammenhangskomponenten mittels einer Übersetzung in eine ausführbare Formulierung gebracht werden.

1 Einleitung

Alle gängigen funktionalen Programmiersprachen bieten Mechanismen, um Werte an Namen zu binden. Für das „Programmieren im Kleinen“ sind das z. B. lokale Definitionen mit „let-in-“ oder „where“-Ausdrücken, für das „Programieren im Großen“ Module oder Strukturen als Sammlung globaler Definitionen. Dabei unterscheiden sich die verschiedenen Sprachen darin, ob Definitionen in beliebiger oder in Abhängigkeitsreihenfolge notiert werden, ob Sichtbarkeiten rekursiv oder linear sind, ob innere Namen äußere verschatten oder überlagern und welche hierarchischen Schachtelungen der verschiedenen Ausdrücke legal sind.

Ebenso gibt es in Form von Tupeln, Records, Objekten oder Datenkonstruktoren Möglichkeiten, eine Menge von Werten zu einer Einheit zusammenzufassen sowie durch Selektionsoperationen die einzelnen Komponenten zu extrahieren.

Wir haben Module dem ersten Aspekt zugeordnet, ebenso können wir sie aber auch als eine Zusammenfassung von Definitionen zu einer Einheit auffassen und sie dem zweiten Aspekt zuordnen. Denn so wie z. B. Records Werte strukturieren, geben Module Programmen eine Struktur und die einzelnen Komponenten eines Moduls können i. d. R. analog zu Selektionsoperatoren außerhalb des Moduls genutzt werden.

An dieser Stelle können wir somit keine klare Trennung zwischen den beiden Aspekten *Bindung von Werten an Namen* und *Zusammenfassen von Werten* ziehen. Dennoch stellen Programmiersprachen unterschiedliche Ausdrucksmittel für die beiden Gesichtspunkte bereit, da Bindungsstrukturen zur Übersetzungszeit analysiert werden, wohingegen Datenstrukturen zur Laufzeit aufgebaut und manipuliert werden. Anders ausgedrückt bedeutet das, dass die Menge der Namen, die in einem Programm verfügbar sind, eine statische Eigenschaft ist und somit zur Übersetzungszeit analysiert werden kann, aber die Menge der möglichen Selektionsoperationen, die ein Programm zur Laufzeit ausführt, nicht a priori vorrausgesagt werden kann.

In [1] wird das Konzept der *Gruppe* vorgeschlagen, um den Bindungs- und den Datenstrukturaspekt in einer gemeinsamen Konstruktion zu erfassen. Gruppen werden dabei co-algebraisch als durch ihre Selektoren (Beobachterfunktionen) bestimmte Objekte aufgefasst, aber nicht formal untersucht. Von dieser Sichtweise weichen wir in diesem Beitrag zugunsten eines pragmatischeren auf effiziente Implementierbarkeit zielenden Standpunkts ab und entwickeln eine formale Semantik für Gruppen sowie notwendige Analysetechniken, um zu einer ausführbaren Formulierung zu gelangen. Zu diesem Zweck definieren wir eine einfache funtionale Programmiersprache GLang, die im Wesentlichen ein um den Gruppenmechanismus erweiterter dynamisch typisierter λ-Kalkül ist.

GLang ist eine strikte Programmiersprache mit Call-by-value-Semantik. Diese Festlegung ist eine Entwurfsentscheidung, da Call-by-value etwas effizienter implementiert werden kann als eine Call-by-need-Semantik und die Auswertungsreihenfolge leichter nachvollzogen werden kann, was etwa die Fehlersuche erleichtert. Neben diesen beiden Punkten spielt auch eine Rolle, dass wir das Konzept der Gruppe zwar in einem rein funktionalen Kontext untersuchen, aber einige Aspekte von generellerer Natur sind und auch auf Sprachen mit Seiteneffekten

zutreffen. Seiteneffekte lassen sich aber mit einer bedarfsgesteuerten Auswertung praktisch nicht kombinieren.

Über diese Entscheidung kann man natürlich trefflich streiten — wir halten an dieser Stelle lediglich fest, dass die Call-by-value-Semantik auf eine Reihe interessanter Fragestellungen führt, um die es in den folgenden Abschnitten gehen wird:

- In Kap. 2 wird die Sprache GLang eingeführt und intuitiv beschrieben. Weiterhin werden einige Idiome gezeigt, wie sich die eingangs benannten und weitere Sprachelemente formulieren lassen. Dieser Abschnitt motiviert auch die Notwendigkeit einer speziellen Abhängigkeitsanalyse.
- Diese Analyse wird im Detail in Kap. 3 entwickelt.
- Aufbauend auf dem Ergebnis der Abhägigkeitsanalyse definieren wir in Kap. 4 eine denotationelle Semantik der Sprache GLang.
- Kapitel 5 vergleicht unseren Ansatz mit ähnlichen Arbeiten.
- Kapitel 6 fasst unsere Ergebnisse zusammen und gibt einen Ausblick auf laufende und zukünftige Arbeiten.

2 Die Sprache GLang

Die Syntax der Sprache GLang ist in Abb. 1 gezeigt. Es gelten die üblichen Klammerkonventionen und Assoziativitäten. Wir nehmen eine abzählbar unendliche Menge x von Variablenbezeichnern sowie primitive Funktionen und Notationen für Ganzzahlen n, Wahrheitswerte b und Strings s an.

$$
\begin{array}{lll}
e & ::= \lambda\ x\ .\ e & \text{— Abstraktion} \\
& \mid\ e\ e & \text{— Applikation} \\
& \mid\ \texttt{IF}\ e\ \texttt{THEN}\ e\ \texttt{ELSE}\ e & \text{— Fallunterscheidung} \\
& \mid\ n \mid b \mid s & \text{— Ganzzahlen, Wahrheitswerte, Strings} \\
& \mid\ x & \text{— Variable} \\
& \mid\ e\ .\ x & \text{— Selektion} \\
& \mid\ \texttt{\{}\ d^*\ \texttt{\}} & \text{— Gruppe} \\
d & ::= x\ \texttt{=}\ e & \text{— Definition}
\end{array}
$$

Abbildung 1. Syntax der Sprache GLang.

2.1 Ein einführendes Beispiel: Listen

Abstraktion, Applikation und Fallunterscheidung haben ihre übliche Bedeutung. Interessanter sind die Bildung von Gruppen und die Selektion.

Eine Gruppe ist eine Menge benannter Definitionen bzw. *Items*.[1] Abbildung 2 zeigt das ubiquitäre Listenbeispiel. Die Gruppe `List` enthält fünf Definitionen

[1] Der Begriff *Item* wird in [1] eingeführt und wir benutzen Definition und Item in diesem Beitrag synonym.

```
1  List = {
2      nil  = {}
3      cons = λx.λxs. { hd=x  tl=xs }
4      head = λxs. xs.hd
5      tail = λxs. xs.tl
6      enum = λn. { enum = λi.
7                      IF i==n THEN nil ELSE cons i (enum (i+1))
8                  }.enum 0
9  }
```

Abbildung 2. Listen in GLang.

`nil`, `cons`, `head`, `tail` und `enum`.[2] Die Gruppenkonstruktion spielt hier die Rolle eines Moduls, das verwandte Definitionen zusammenfasst. Die Reihenfolge der Definitionen innerhalb einer Gruppe spielt keine Rolle, in diesem Sinne können die umschließenden Klammern „{“ und „}“ als Mengenklammern gelesen werden.

Die beiden Konstruktoren `nil` und `cons` implementieren Listenelemente ebenfalls als Gruppen; `nil` ist die leere Gruppe, die keinerlei Definitionen enthält, und `cons` liefert eine Gruppe, die den Listenkopf `hd` und die Restliste `tl` enthält. Hier treten Gruppen also als Datenstrukturen auf, die zur Laufzeit aufgebaut und manipuliert werden.

Die beiden Funktionen `head` und `tail` selektieren das erste Element bzw. die Restliste ihres Arguments mittels des Selektionsoperators „`.`“ und abstrahieren damit von den konkreten Selektoren `.hd` und `.tl`, die in der Implementierung von `cons` verwendet werden.

Die Funktion `enum` bildet eine Liste der Zahlen von `0` bis `n`. An ihrer Definition sind mehrere Aspekte zu erkennen:

- Auf der rechten Seite einer Definition können die in der umschließenden Gruppe definierten Variablen benutzt werden, in diesem Fall sind das `nil` und `cons` aus der Gruppe `List`.
- Der Rumpf der Funktion `enum` definiert eine anonyme Gruppe, die das Item `enum` enthält. Wir nennen eine Gruppe anonym, wenn sie nicht der oberste Knoten im Syntaxbaum auf der rechten Seite einer Definition ist.
- In der Definition von `enum` in der anonymen Gruppe wird auf der rechten Seite im `ELSE`-Zweig die Variable `enum` verwendet. Aufgrund der lexikalischen Sichtbarkeit bezieht sich diese Verwendung auf die innerste sichtbare Definition, also auf die Funktion `enum` in der anonymen Gruppe und nicht auf `List.enum`. Es handelt sich also um eine rekursive Definition.

[2] Für eine benutzbare Listenimplementierung fehlen hier noch die Diskriminatoren `isNil` und `isCons`. Wir benötigen dazu den primitiven Gruppen-Diskriminator `DEFINES`, mit dem getestet werden kann, ob eine Gruppe genau die gegebenen Variablen definiert. Da `DEFINES` ein rein dynamischer Test ist, brigt er keine zusätzliche Schwierigkeit, so dass wir ihn in diesem Beitrag ignorieren.

– Im Rumpf von `List.enum` rufen wir durch den Selektor `.enum` diese rekursive Funktion mit dem Startargument `0` direkt auf. Die anonyme Gruppe sowie die direkte Selektion spielen hier also die Rolle eines lokalen rekursiven „let-in“- oder „where“-Ausdrucks.

2.2 Funktional-objektorientierte Programmierung

Unser zweites Beispiel zeigt, wie Gruppen genutzt werden können, um in einem eher objektorientierten Stil zu programmieren.

Abbildung 3 zeigt einen Eval-Apply-Interpreter mit Umgebungen für den ungetypten λ-Kalkül. Ein Term ist entweder eine Variable `Var`, Abstraktion `Abs` oder eine Applikation `App`. Diese Termkonstruktoren sind der „objects-as-closures“-Implementierung [2] recht ähnlich, allerdings werten sie nicht zu einer Closure, die eine Nachricht erwartet, sondern zu einer Gruppe aus. Diese Gruppen enthalten eine Auswertungsfunktion `eval`, die eine Umgebung Γ, die freien Variablen des Terms Werte zuordnet, nimmt und den Wert des Terms berechnet. Dabei wird ggf. auf die Auswertungsfunktion der Teilterme zurückgegriffen.

Beim Auswerten einer Variable wird ihr Wert aus der Umgebung Γ ermittelt (Zeile 2). Das Auswerten eine Abstraktion besteht lediglich aus dem Bilden einer Closure als Gruppe mit der abstrahierten Variable `var`, dem Funktionsrumpf `body` und der lexikalischen Umgebung `env` (Zeile 3).

Bei der Reduktion einer Applikation werden zunächst Funktions- und Argumentterm ausgewertet (Zeile 4). Im zweiten Schritt wird mittels `apply` der Rumpf der Closure `f.body` in der um das Argument angereicherten Umgebung ausgewertet (Zeilen 5-6).

```
1  Term = {
2    Var    = λx. { eval = λΓ. Γ.lookup x }
3    Abs    = λx.λt. { eval = λΓ. { var=x  body=t  env=Γ } }
4    App    = λt1.λt2. { eval  = λΓ. apply (t1.eval Γ) (t2.eval Γ)
5                        apply = λf.λa.
6                          f.body.eval (f.env.add f.var a) }
7  }
```

Abbildung 3. Ein Eval-Apply-Interpreter im funktional-objektorientiertem Stil.

Die Implementierung der Umgebung für den Eval-Apply-Interpreter ist in Abb. 4 gezeigt. Eine Umgebung wird dabei als Liste von Paaren `{var=x val=v}` unter Zuhilfenahme der Listenimplementierung aus Abb. 2 umgesetzt und stellt die beiden Funktionen `add`, um eine Bindung hinzuzufügen, und `lookup`, um den Wert einer Variablen zu ermitteln, bereit. Der Konstruktor `Env` erzeugt eine Umgebung, die initial alle in `bdgs` aufgeführten Variablenbindungen enthält.

Im Unterschied zu Termen müssen währen der Auswertung neue Umgebungen erzeugt werden. Dies wird in `add` dadurch realisiert, dass mittels `Env` eine

neue Umgebung erstellt wird, die alle vorherigen Bindung `bdgs` sowie die neue Bindung `Bdg x v` enthält.

```
1  Env = λbdgs. {
2    Bdg    = λx.λv. { var=x  val=v }
3    add    = λx.λv. Env (List.cons (Bdg x v) bdgs)
4    lookup = λx. { find = λ bdgs. {
5                               bdg = List.head bdgs
6                               val = IF bdg.var==x
7                                       THEN bdg.val
8                                       ELSE find (List.tail bdgs)
9                           }.val
10               }.find bdgs
11 }
```

Abbildung 4. Umgebung für den Eval-Apply-Interpreter.

Wir können an dieser Stelle folgende Beobachtungen festhalten:

- Objekte werden als Gruppen realisiert. Methoden sind Items in einer Gruppe.
- Konstruktoren aus objektorientierten Programmiersprachen können mittels Gruppen als Funktionen, die Gruppen liefern, erfasst werden.
- Durch λ abstrahierte Variablen in der Konstruktordefinition spielen die Rolle von Attributen. Auf sie kann nicht von Außen zugegriffen werden. Sie können aber auch als Items modelliert werden, so dass sie von außen selektiert werden können.
- Das Aufrufen einer Methode bzw. das Senden einer Nachricht korrespondiert mit der Selektion einer Funktion aus einer Gruppe.
- Die Bindungsregeln von Gruppen sorgen dafür, dass solch eine „Methode“ Zugriff auf andere „Methoden“ und „Attribute“ des Objekts hat. Eine spezielle Referenz wie `this` in Java oder `self` in Smalltalk gibt es nicht.

Erweiterbarkeit Wie in anderen objektorientieren Sprachen können wir unser System leicht um neue Daten erweitern,[3] in dem wir weitere Termkonstruktoren hinzufügen. Abbildung 5 zeigt drei Termkonstruktoren `Zero`, `Succ` und `IfZero`, die den Interpreter um natürliche Zahlen und eine einfache Fallunterscheidung erweitern. Die bereits vorhandenen Definitionen bleiben von der Erweiterung unbeeinträchtigt und alle Termarten können beliebig kombiniert werden.

[3] Die Erweiterung um neue Funktionalität ist allerdings ebenso unhandlich, da alle Konstruktoren um eine neue Funktion angereichert werden müssen.

```
1    Zero   = { eval = λ Γ. 0 }
2    Succ   = λt. { eval = λΓ. add (t.eval Γ) 1 }
3    IfZero = λt1.λt2.λt3. { eval = λΓ. IF (t1.eval Γ)==0
4                                       THEN t2.eval Γ
5                                       ELSE t3.eval Γ }
```

Abbildung 5. Erweiterung des Eval-Apply-Interpreters um natürliche Zahlen.

2.3 Rekursion über Gruppengrenzen

In Abb. 2 haben wir anhand der Funktion `enum` bereits gesehen, dass Definitionen rekursiv sein können. Dabei sind gegenseitig rekursive Funktionsdefinitionen möglich. Abbildung 6 zeigt ein solches Beispiel, bei dem die Rekursion sogar Gruppen-übergreifend ist.

```
1  { E = { even    = λn. IF n==0 THEN true ELSE O.odd (n-1)
2          is2even = even 2 }
3    O = { odd     = λn. IF n==0 THEN false ELSE E.even (n-1)
4          is2odd = odd 2 }
5  }
```

Abbildung 6. Gruppenübergreifende gegenseitig rekursive Funktionen.

In diesem Beispiel werden die zwei Funktionen `E.even` und `O.odd` definiert, die bestimmen, ob eine Zahl gerade oder ungerade ist. Dabei macht `E.even` von `O.odd` Gebrauch und umgekehrt. Gleichzeitig wird `even` noch in `E.is2even` und `odd` noch in `O.is2odd` verwendet.

Wie bereits in Abschn. 1 erwähnt, hat GLang eine Call-by-value-Semantik. Das bedeutet insb., dass eine Variable an einen Wert gebunden sein muss, bevor sie verwendet wird. Da Definitionen Werte an Variablen binden, müssen die rechten Seiten in Abhängigkeitsreihenfolge ausgewertet werden. Für die Definitionen der Gruppen `E` und `O` bedeutet dies, dass jeweils `even` bzw. `odd` vor `is2even` bzw. `is2odd` ausgewertet werden. Allerdings muss sowohl `E` vor `O` gebildet werden, da `O.is2odd` von `O.odd` abhängt, das `E.even` benötigt, als auch `O` vor `E`, da `E.is2even` von `E.even` abhängt, das `O.odd` benötigt. `E` und `O` sind in einem Abhängigkeitszyklus. In einer Call-by-value-Semantik müssen die an einem Zyklus beteiligten rechten Seiten aber Funktionen sein, da sonst die Auswertung nicht terminiert.

Wir lösen das Dilemma, indem wir die Abhängigkeiten über Gruppengrenzen betrachten und Selektorketten berücksichtigen. Eine abhängigkeitskompatible Auswertungsreihenfolge ist z. B. `E.even`, `O.odd`, `E.is2even`, `E`, `O.is2odd`, `O`.

Wir lösen durch die Berücksichtigung der Selektorketten die hierarchische Struktur auf, ermitteln die Abhängigkeiten und berechnen eine kompatible Aus-

wertungsreihenfolge. Leider ist dieses Verfahren i. A. nicht anwendbar, wenn Gruppen als Datenstrukturen verwendet werden, wie der nächste Abschnitt zeigt.

2.4 Ein unentscheidbares Problem

Betrachten wir das, zugegebenermassen artifizielle, Beispiel in Abb. 7. Wenn wir, wie im vorherigen Abschnitt, beschrieben, generell die Selektorketten bei der Abhängigkeitsermittlung berücksichtigen, müssen wir feststellen, ob der Name `output.tl.tl.hd` definiert wird. Wie in Abb. 7 plakativ angedeutet, existiert das dritte Listenelement, wenn die Eingabe einem Prädikat `isValid` nicht genügt, was i. A. nicht entscheidbar ist.

```
1  λinput. {
2    seq    = IF isValid input THEN List.nil
3                              ELSE List.enum 3
4    output = seq.tl.tl.hd
5  }
```

Abbildung 7. Ein problematisches Beispiel (unter Nutzung von `List` aus Abb. 2).

Wir stellen allerdings fest, dass es in diesem Falle genügt, zu erkennen, dass `output` von `seq` abhängt, wodurch die Auswertungsreihenfolge `seq`, `output` lautet. Falls nun die Liste `seq` nicht dreielementig ist, gibt es einen Laufzeitfehler, was bei einer fehlgeschlagenen Listenselektion akzeptabel und üblich ist.

In manchen Fällen muss also die Selektorkette berücksichtigt werden, in anderen darf sie nicht (oder nur zu Teilen) in die Bestimmung der Abhängigkeiten einfließen. Die genauen Kriterien und die Ermittlung der Abhängigkeiten, die den beiden Problemfällen aus diesem und dem vorherigen Abschnitt gerecht wird, ist der Gegenstand des folgenden Kapitels.

3 Abhängigkeitsanalyse und Transformation

In diesem Abschnitt stellen wir die Abhängigkeitsanalyse vor. Als erstes werden die für die Analyse benötigten Begriffe definiert. Im Anschluß daran wird die Analyse an einem Beispiel vorgestellt und anschließend werden die Funktionen definiert.

3.1 Indizierte Ausdrücke

Um die Darstellung der Abhängigkeitsanalyse zu vereinfachen, gehen wir von der Syntax aus Abb. 1 zu der abgewandelten Darstellung aus Abb. 8 über. Dabei trennen wir die Gruppen von den restlichen Ausdrücken, indem wir das

$$
\begin{array}{rcl}
E & ::= & G \mid T \\
T & ::= & [\lambda\ x\ .\ E]^{\ell} \\
 & \mid & [E\ E]^{\ell} \\
 & \mid & [\texttt{IF}\ E\ \texttt{THEN}\ E\ \texttt{ELSE}\ E]^{\ell} \\
 & \mid & n^{\ell} \mid b^{\ell} \mid s^{\ell} \\
 & \mid & x^{\ell} \\
 & \mid & [E\ .\ x]^{\ell} \\
G & ::= & \{\ D^{*}\ \}^{\ell} \\
D & ::= & x = G \mid x = T
\end{array}
$$

Abbildung 8. Syntax indizierter Ausdrücke.

ursprüngliche e in G und T aufspalten. Ein GLang-Ausdruck E kann dann entweder ein G oder T sein. Weiterhin annotieren wir alle Ausdrücken $G \cup T$ mit einem eindeutigen Index aus der Menge ℓ und nutzen ggf. eckige Klammern, um kenntlich zu machen, auf welchen Ausdruck sich ein Index bezieht. Die Inverse der Indizierung speichern wir in einer Abbildung $EXP : \ell \hookrightarrow G \cup T$.

Wir nutzen die Indizes, um zusätzliche Informationen mit dem jeweiligen Ausdruck zu verknüpfen.

Die Transformation von e nach E und die Indizierung der Ausdrücke lassen wir aus, sie birgt keinerlei überraschende Erkenntnisse.

3.2 Definitionen

Wir nutzen im weiteren Verlauf des Artikels die folgenden Begriffe:

Selektor / Name Ein *Selektor* ist ein Bezeichner x mit vorangestelltem Punkt. Eine *Selektorkette* ist ein Folge von Selektoren.

Ein *Name* ist eine Variable gefolgt von einer (potentiell leeren) Selektorkette. Wir bezeichnen die Menge aller Namen mit N.

Freie und verfügbare Namen Wir ordnen jedem Ausdruck T, G eines Programms mittels des Index ℓ zwei Mengen zu:

$FN : \ell \hookrightarrow \mathbb{P}N$ Menge der freien Namen

$AVG : \ell \hookrightarrow \mathbb{P}x$ Menge der gruppengebundenen verfügbaren Variablen

Freie Namen erweitern das Konzept der freien Variablen aus dem λ-Kalkül auf Namen, indem ein Name genau dann frei ist, wenn seine führende Variable frei ist. Die Funktion $\mathcal{F}$ aus Abb. 9 ordnet jedem Ausdruck eines Programms seine freien Namen zu.

Der Unterschied zur üblichen Menge der freien Variablen ist lediglich die Gleichung $(*)$, wo der gesamte Name $x_1 . \cdots . x_m$ statt nur der Variablen x_1 aufgenommen wird. Die Abbildung FN ergibt sich nun durch die Komposition $FN = \mathcal{F} \circ EXP$.

$\mathcal{F} : \; G \cup T \to \mathbb{P}N$		
$\mathcal{F}[\![\lambda x\,.\,E]\!]$	$= \mathcal{F}[\![E]\!] \ominus \{x\}$	
$\mathcal{F}[\![E_1\ E_2]\!]$	$= \mathcal{F}[\![E_1]\!] \cup \mathcal{F}[\![E_2]\!]$	
$\mathcal{F}[\![\texttt{IF}\ E_1\ \texttt{THEN}\ E_2\ \texttt{ELSE}\ E_3]\!]$	$= \mathcal{F}[\![E_1]\!] \cup \mathcal{F}[\![E_2]\!] \cup \mathcal{F}[\![E_3]\!]$	
$\mathcal{F}[\![x_1\,.\,\cdots\,.\,x_m]\!]$	$= \{x_1\,.\,\cdots\,.\,x_m\}$	(9.4)
$\mathcal{F}[\![E\,.\,x]\!]$	$= \mathcal{F}[\![E]\!]$	
$\mathcal{F}[\![x]\!]$	$= \{x\}$	
$\mathcal{F}[\![\alpha]\!]$	$= \emptyset$	für $\alpha \in \{n, b, s\}$
$\mathcal{F}[\![\{\, x_i = E_i{}^{i\in 1..m} \,\}]\!]$	$= \left\{ N \mid N = x_1\,.\,\cdots\,.\,x_p \wedge N \in \bigcup_{i\in 1..m} \mathcal{F}[\![E_i]\!] \wedge x_1 \notin \{x_i{}^{i\in 1..m}\} \right\}$	

Abbildung 9. Funktion zu Berechnung freier Namen.

$\mathcal{A}^\alpha : \; \alpha \to \mathbb{P}x \to (\ell \hookrightarrow \mathbb{P}x)$	für $\alpha \in \{E, T, G, D\}$	
$\mathcal{A}^E[\![T]\!]\Gamma$	$= \mathcal{A}^T[\![T]\!]\Gamma$	
$\mathcal{A}^E[\![G]\!]\Gamma$	$= \mathcal{A}^G[\![G]\!]\emptyset$	(†)
$\mathcal{A}^T[\![[\lambda x\,.\,E]^\ell]\!]\Gamma$	$= \{\ell \mapsto \Gamma\} \cup (\mathcal{A}^E[\![E]\!]\Gamma \setminus \{x\})$	
$\mathcal{A}^T[\![[E_1\ E_2]^\ell]\!]\Gamma$	$= \{\ell \mapsto \Gamma\} \cup \mathcal{A}^E[\![E_1]\!]\Gamma \cup \mathcal{A}^E[\![E_2]\!]\Gamma$	
$\mathcal{A}^T[\![[\texttt{IF}\ E_1\ E_2\ E_3]^\ell]\!]\Gamma$	$= \{\ell \mapsto \Gamma\} \cup \mathcal{A}^E[\![E_1]\!]\Gamma \cup \mathcal{A}^E[\![E_2]\!]\Gamma \cup \mathcal{A}^E[\![E_3]\!]\Gamma$	
$\mathcal{A}^T[\![[E\,.\,x]^\ell]\!]\Gamma$	$= \{\ell \mapsto \Gamma\} \cup \mathcal{A}^E[\![E]\!]\Gamma$	
$\mathcal{A}^T[\![\alpha^\ell]\!]\Gamma$	$= \{\ell \mapsto \Gamma\}$	für $\alpha \in \{x, n, b, s\}$
$\mathcal{A}^G[\![\{\, D_i{}^{i\in 1..m} \,\}^\ell]\!]\Gamma$	$= \{\ell \mapsto \Gamma\} \cup \left(\bigcup_{i\in 1..m} \mathcal{A}^D[\![D_i]\!](\Gamma \cup \{x_i{}^{i\in 1..m}\}) \right)$ mit $D_i = x_i = T_i$ oder $D_i = x_i = G_i$	
$\mathcal{A}^D[\![x = T]\!]\Gamma$	$= \mathcal{A}^T[\![T]\!]\Gamma$	
$\mathcal{A}^D[\![x = G]\!]\Gamma$	$= \mathcal{A}^G[\![G]\!]\Gamma$	(‡)

Abbildung 10. Berechnung der gruppengebundenen verfügbaren Variablen.

Verfügbare Variablen sind die Variablen, die ein Ausdruck T durch den äußeren Kontext zur Verfügung hat. Wir sind insb. an den Variablen interessiert, die in einer ununterbrochenen Gruppenhierarchie gebunden werden, deren Syntaxbaum also nur die Nichtterminale G und D aus Abb. 8 enthält. Die Menge dieser Variablen nennen wir *gruppengebundene verfügbare Variablen* und speichern sie in der Abbildung AVG, die durch die Funktion $\mathcal{A}^E$ im Zusammenspiel mit $\mathcal{A}^T$, $\mathcal{A}^G$, $\mathcal{A}^D$ aus Abb. 10 berechnet wird.

Das zweite Argument der Funktion $\mathcal{A}^E$ ist die Menge der gruppengebundenen verfügbaren Namen aus dem äußeren Kontexts eines Ausdrucks E. Die Abbildung AVG für ein Programm E ist also definiert als $AVG = \mathcal{A}^E[\![E]\!]\emptyset$, da der äußere Kontext leer ist.

In Gleichung (†) wird die Menge der gruppengebundenen verfügbaren Variablen geleert. Hier beginnt also eine neue Gruppenhierarchie. Dagegen wird Γ in Gleichung (‡) beibehalten – die in der aktuell betrachteten Gruppenhierarchie verfügbaren Variablen wachsen. Die Möglichkeit, die Funktionen $\mathcal{A}^\alpha$ auf diese

einfache Weise rein syntaktisch zu definieren, ist einer der Gründe, Gruppenausdrücke von den anderen Ausdrücken zu separieren.

Wir illustrieren die Abbildung AVG am Beispiel in Abb. 11. Die gruppengebundenen verfügbaren Variablen für die Ausdrücke 1–4 werden in den durch { und } gekennzeichnetet Gruppen gebunden. In Ausdruck 1 und 2 ist dieselbe Menge an gruppengebundenen Variablen verfügbar. In Ausdruck 3 ist das `h` nicht mehr in einer Gruppe, sondern durch eine λ-Abstraktion gebunden. Ausdruck 4 hat die Variablen `h` und `f` nicht und stattdessen `m` verfügbar, da Ausdruck 4 in einem anderen Zweig der Hierarchie steht. Allen vier Ausdrücken sind natürlich auch die Variablen `k`, `c` sowie `a` verfügbar, aber die Ersteren stammen nicht aus der umschließenden ununterbrochenen Gruppenhierarchie und Letzterer ist durch eine λ-Abstraktion gebunden.

```
1  { ···
2    k = { x = 1 }
3    c = λa. { A = { h = a[1]
4                    f = [λh. [k.x + h][3]][2]
5                  }
6              f = { m = [λy. y][4] }
7            }
8  ··· }
9
```

$AVG(1) = \{\texttt{A},\texttt{h},\texttt{f},\texttt{t}\}$
$AVG(2) = \{\texttt{A},\texttt{h},\texttt{f},\texttt{t}\}$
$AVG(3) = \{\texttt{A},\texttt{f},\texttt{t}\}$
$AVG(4) = \{\texttt{A},\texttt{t},\texttt{m}\}$

Abbildung 11. Beispiel zur Menge der gruppengebundenen verfügbaren Namen.

3.3 Analyse

Mit Hilfe einer speziellen Abhängigkeitsanalyse wird ein Ausdruck E in eine neue Zwischenrepräsentation (siehe Abb. 17) transformiert. In dieser Darstellung ist die Auswertungsreihenfolge explizit. In Abschnitt 4 wird für diese Repräsentation eine denotationelle Semantik angegeben. Die Abhängigkeitsanalyse muss für jede ununterbrochenen Gruppenhierarchie durchgeführt werden. Dabei durchläuft die Analyse vier Phasen:

1. Der Gruppenbaum wird aufgebaut.
2. Die Abhängigkeitskanten werden ermittelt.
3. Mit Hilfe des Algorithmus von Sharir [3] werden die starken Zusammenhangskomponenten (SCC) des Abhängigkeitsgraphen berechnet, um die Abhängigkeitsreihenfolge zu ermitteln.
4. Transformation der starken Zusammenhangskomponenten in einen Baum.

Im Folgenden werden diese vier Schritte an einem Beispiel intuitiv erläutert. Im Anschluss daran werden die Schritte definiert.

Beispiel Zur Illustration des Algorithmus verwenden wir das bereits bekannte Beispiel (siehe Abb. 12) verschränkt rekursiver Gruppen, jetzt erweitert um die Indizes.

```
1  { E = { even    = [λn. IF n==0 THEN true ELSE O.odd (n-1)]4
2          is2even = [even 2]5 }2
3    O = { odd     = [λn. IF n==0 THEN false ELSE E.even (n-1)]6
4          is2odd = [odd 2]7 }3
5  }1
```

Abbildung 12. Gruppenübergreifende gegenseitig rekursive Funktionen mit Indizes.

Im ersten Schritt wird für diese ununterbrochene Gruppenhierarchie ein Gruppenbaum aufgebaut. Dieser ist in Abb. 13 dargestellt. In jedem Knoten wird der vollständige Name innerhalb der Gruppenhierarchie und der Index der rechten Seite gespeichert. Die Wurzel jedes Gruppenbaums hat den Namen $\overline{root}$ und den Index der äußersten Gruppe, in diesem Fall also 1. Die Definitionen der Gruppen werden als Kinder des Knotens eingefügt.

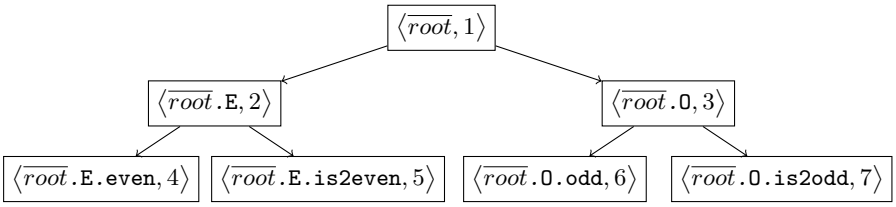

Abbildung 13. Gruppenbaum für das Beispiel verschränkt rekursiver Gruppen.

Im zweiten Schritt müssen alle Abhängigkeitskanten ermittelt werden. Dazu werden alle Blätter betrachtet. Dies sind im Beispiel die Knoten mit den Indizes 4–7. Zunächst muss die Menge der gruppengebundenen verwendeten Namen (*UNG*) ermittelt werden. Dies sind alle freien Namen, deren Variable in der Menge der gruppengebundenen verfügbaren Variablen ist. In Abb. 14 sind diese drei Menge für jeden Index berechnet. In diesem Fall entspricht die Menge der gruppengebundenen verfügbaren Namen genau der Menge der freien Namen, das ist aber i. A. nicht der Fall.

Der Knoten, der diesen Namen repräsentiert, muss nun gefunden werden. Dazu wird ausgehend vom betrachteten Blatt der Baum nach oben durchsucht, um die Variable des Namens zu finden. Betrachten wir dazu das Blatt mit dem Index 4 und dem gruppengebundenen Namen `O.odd`. Der direkte Elternknoten hat kein Kind mit der Definition `O`. Somit wird der nächste Elternknoten betrach-

Index	*FN*	*AVG*	*UNG*
4	`O.odd`	`O`, `even`, `is2even`	`O.odd`
5	`even`	`O`, `even`, `is2even`	`even`
6	`E.even`	`E`, `odd`, `is2odd`	`E.even`
7	`odd`	`E`, `odd`, `is2odd`	`odd`

Abbildung 14. Zuordnung der Indizes zu gruppengebundenen verwendeten Namen.

tet. Dieser enthält ein solches Kind, nämlich den Knoten mit dem Index 3. Nun wird ausgehen von diesem Knoten die Selektorkette betrachtet und jeweils ein passendes Kind gesucht. In diesem Fall ist der nächste Selektor `.odd`. Das entsprechende Kind ist der Knoten mit Index 6. Da kein weiterer Selektor vorhanden ist, ist dies auch der gesuchte Knoten. Es wird also eine Abhängigkeitskante von 4 zu 6 eingefügt. Zusätzlich werden auch Kanten zu allen Vorfahren von 6, die keine Vorfahren von 4 sind eingefügt. Dies ist in diesem Fall nur der Knoten 3. Diese zusätzlichen Kanten sind nötig, damit die Abhängigkeiten zwischen den beiden Gruppe `E` und `O` berücksichtigt werden.

Für jeden gruppengebundenen verwendeten Namen müssen diese Kanten in den Graphen eingefügt werden. Der entsprechende Graph ist für unser Beispiel in Abb. 15 dargestellt, wobei alle Abhängigkeitskanten gestrichelt sind.

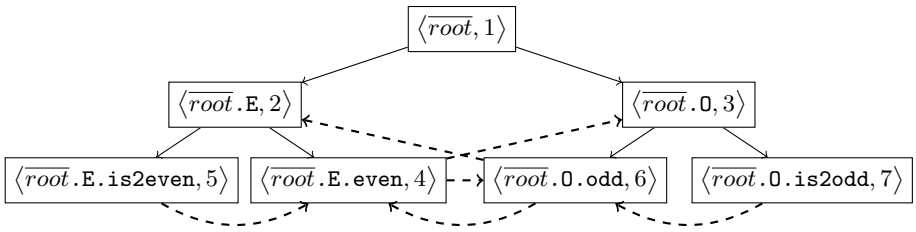

Abbildung 15. Abhängigkeitsgraph für das Beispiel verschränkt rekursiver Gruppen aus Abb. 12.

Auf diesen Graphen wird der Algorithmus von Sharir zur Bestimmung der starken Zusammenhangskomponenten in topologischer Sortierung angewendet. Jede Zusammenhangskomponente ist bei diesem Algorithmus ein Baum. Uns interessieren nur die in dem Baum enthaltenen Knoten. Somit können wir den Baum als Menge von Knoten auffassen. Somit ist das Ergebnis dieses Algorithmus eine Liste von Knotenmengen. In unserem Beispiel stehen die Knoten 2–6 in einer Zusammenhangskomponente. Somit erhalten wir die folgende Liste:

$$\{\langle \overline{root}\texttt{.E}, 2\rangle, \langle \overline{root}\texttt{.O}, 3\rangle, \langle \overline{root}\texttt{.E.even}, 4\rangle \langle \overline{root}\texttt{.E.is2even}, 5\rangle, \langle \overline{root}\texttt{.O.odd}, 6\rangle, \langle \overline{root}\texttt{.O.is2odd}, 7\rangle\}, \{\langle \overline{root}, 1\rangle\}$$

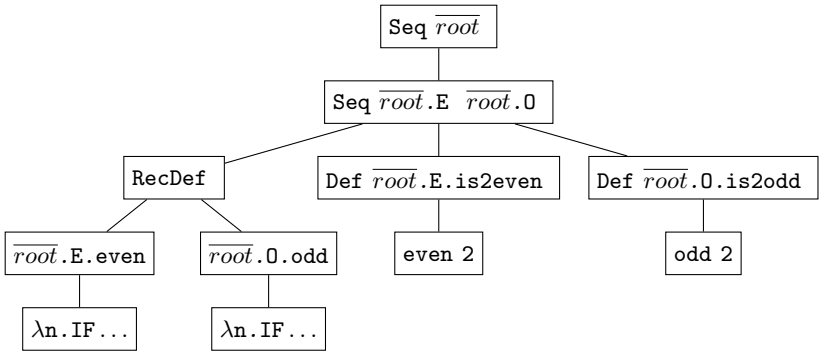

Abbildung 16. Ergebnis der Abhängigkeitsanalyse für verschränkt rekursive Gruppen.

$$
\begin{array}{lll}
A & ::= & C \mid S \\
S & ::= & [\lambda\ x\ .\ A]^{\ell} \mid\ x^{\ell} \mid\ [A\ .\ x]^{\ell} \mid\ \ldots \\
C & ::= & \texttt{Def}\ N\ S \mid\ \texttt{RecDef}\ (N\ S)^{+} \mid\ \texttt{Seq}\ N^{+}\ C^{*}
\end{array}
$$

Abbildung 17. Syntax zur Beschreibung der Auswertungsreihenfolge.

Diese besagt, dass zuerst die Gruppen `E` und `O` gemeinsam berechnet werden müssen und anschließend kann die äußere Gruppe erstellt werden.

Mit Hilfe dieser Liste wird die Gruppe in eine neue baumförmige Zwischenrepräsentation umgewandelt. Dieser neue Baum enthält die Information über die Auswertungsreihenfolge. Für unser Beispiel ergibt sich die Repräsentation in Abb. 16. Die Gruppen `E` und `O` werden in einem Sequenz-Knoten und die beiden verschränkt rekursiven Funktionen `even` und `odd` werden in einer rekursiven Definition zusammengefasst. Die anderen beiden Definitionen sind ebenfalls Kinder des Sequenzknotens.

Diese Zwischenrepräsentation ist das Ergebnis der Abhängigkeitsanalyse. Im Folgenden werden die im Beispiel vorgestellten Funktionen definiert.

Transformation Die Funktion $\mathcal{S}^E$ (zusammen mit $\mathcal{S}^G$, $\mathcal{S}^T$) aus Abb. 18 transformiert einen Ausdruck E in einen Ausdruck A (siehe Abb. 17). S entspricht weitgehend der abstrakten Syntax für T, nur die Nicht-Terminale E sind durch A ersetzt, C repräsentiert die Gruppen.

Definitionen werden unterteilt in nicht rekursiv (`Def`) und rekursiv (`RecDef`), wobei verschränkt rekursive Definitionen zusammengefasst werden. Alle Gruppen werden zu `Seq` transformiert. Verschränkt rekursive Gruppen werden in einem `Seq` zusammengefasst. Die Definitionen der Gruppe bzw. Gruppen werden in Abhängigkeitsreihenfolge als Kinder des Knotens eingefügt.

Die Transformation geht rekursiv über die abstrakte Syntax. Jede ununterbrochene Gruppenhierarchie wird als Ganzes analysiert und transformiert. Die

$\mathcal{S}^E : E \to A$ $\quad \mathcal{S}^G : T \to S$ $\quad \mathcal{S}^T : G \to C$	
$\mathcal{S}^E[\![T]\!]$	$= \mathcal{S}^T[\![T]\!]$
$\mathcal{S}^E[\![G]\!]$	$= \mathcal{S}^G[\![G]\!]$
$\mathcal{S}^T[\![\lambda x\,.\,E]\!]$	$= \lambda x\,.\,\mathcal{S}^E[\![E]\!]$
$\mathcal{S}^T[\![E_1\ E_2]\!]$	$= \mathcal{S}^E[\![E_1]\!]\ \mathcal{S}^E[\![E_2]\!]$
$\mathcal{S}^T[\![\texttt{IF}\ E_1\ \texttt{THEN}\ E_2\ \texttt{ELSE}\ E_3]\!]$	$= \texttt{IF}\ \mathcal{S}^E[\![E_1]\!]\ \texttt{THEN}\ \mathcal{S}^E[\![E_2]\!]\ \texttt{ELSE}\ \mathcal{S}^E[\![E_3]\!]$
$\mathcal{S}^T[\![E\,.\,x]\!]$	$= \mathcal{S}^E[\![E]\!]\,.\,x$
$\mathcal{S}^T[\![\alpha]\!]$	$= \alpha \quad$ für $\alpha \in \{x, n, b, s\}$
$\mathcal{S}^G[\![G^\ell]\!]$	$= \mathcal{T}[\![G]\!]\overline{root}\ scc\ \gamma$
	mit $\quad scc = \mathsf{calcScc}\,\gamma$
	$\gamma = \mathsf{depGraph}\,\gamma_1$
	$\gamma_1 = \mathsf{groupTree}\,G\ \langle\overline{root}, \ell\rangle\ \overline{root}$

Abbildung 18. Transformation in einen Baum mit expliziter Auswertungsreihenfolge.

vier Schritte der Analyse entsprechen den vier Schritten in der Funktion $\mathcal{S}^G$ und werden in den folgenden Abschnitten definiert.

Gruppenbaum (groupTree) Die Gruppenhierarchie wird als Baum repräsentiert. Die Wurzel dieses Baumes repäsentiert die äußerste Gruppe der Hierarchie und wird im Folgenden $\overline{root}$ genannt. Alle Definitionen einer Gruppe sind die Kinder des Gruppenknotens. In jedem Knoten wird der vollständige Name dieses Selektors und der Index der rechten Seite gespeichert. Die Blätter repräsentieren die Definitionen, deren rechte Seite keine Gruppe ist. Die Funktion groupTree erstellt für einen Gruppenausdruck den Gruppenbaum. Da der Baum in einem späteren Schritt zu einem Graphen γ erweitert wird, wird der Baum durch das Tripel $\langle\Upsilon, \varepsilon, \upsilon\rangle$ repräsentiert, wobei Υ die Menge der Knoten, ε die Menge der Kanten und υ der Wurzelknoten ist. $\mathsf{groupTree} : \ E \to \upsilon \to N \to \gamma$

$$\begin{aligned}
\mathsf{groupTree}[\![\{\, x_i = E_i^{\ell_i}{}^{i\in 1..m} \,\}]\!]\ \upsilon\ N &= \left\langle \{\upsilon\} \cup \bigcup_{i\in 1..m} \Upsilon_i, \varepsilon \cup \bigcup_{i\in 1..m} \varepsilon_i, \upsilon \right\rangle \\
&\text{mit} \quad \langle \Upsilon_i, \varepsilon_i, \upsilon_i\rangle = \mathsf{groupTree}[\![E_i]\!]\ \upsilon_i\ (N\,.\,x_i) \\
&\qquad \upsilon_i = \langle N\,.\,x_i, \ell_i\rangle \\
&\qquad \varepsilon = \{\langle \upsilon, \upsilon_i\rangle \mid i \in 1..m\} \\
\mathsf{groupTree}[\![T]\!]\ \upsilon\ N &= \langle \{\upsilon\}, \emptyset, \upsilon\rangle
\end{aligned}$$

Aus jeder rechten Seite einer Gruppe wird ein Teilbaum erstellt. Diese Teilbäume sind die Kinder des Knotens, der die Gruppe repräsentiert. Jeder Ausdruck, der keine Gruppe ist, ist als Blatt in dem Baum repräsentiert.

Abhängigkeitsgraph (depGraph) Um aus dem Gruppenbaum einen Abhängigkeitsgraphen zu erhalten, müssen für alle Blätter die Abhängigkeitskanten bestimmt werden. Dazu wird für die rechte Seite die Menge aller gruppengebundenen verwendeten Namen (UNG) ermittelt:

$$UNG(\ell) \qquad = \{N \mid N = x_1 \,.\, \cdots \,.\, x_m \wedge N \in FN(\ell) \wedge x_1 \in AVG(\ell)\}$$

$$\mathsf{allDepEdges}\,\gamma = \bigcup_{\langle N,\ell\rangle \in \mathsf{leaves}\,\gamma} \left(\bigcup_{N' \in UNG(\ell)} \mathsf{depEdges}\,\gamma\ \langle N, \ell\rangle\ N' \right)$$

Für jede dieser Variablen werden Abhängigkeitskanten in den Graphen eingefügt. Zum einen wird immer eine Kante von dem Blatt zu dem Knoten, der den Namen im Baum repräsentiert, eingefügt. Dazu muss der Zielknoten der Kante ermittelt werden. Außerdem wird zu jedem Vorfahren des Zielknotens, der kein Vorfahre des betrachteten Blattes ist, eine Kante eingefügt. Im ersten Schritt wird dazu die Variable des Namens im Baum gesucht und im zweiten Schritt wird der statische Anteil der Selektorkette ermittelt. Im Anschluß daran werden zusätzliche Kanten zu den Vorfahren ermittelt. $\mathsf{depEdges} :\ \gamma \to \upsilon \to N \to \mathbb{P}E$

$$\mathsf{depEdges}\,\gamma\,\upsilon\,(x\,.\,N) = \begin{cases} \mathit{Error}, & \text{falls } \upsilon' = \mathit{Error} \\ \{\langle \upsilon, \upsilon'\rangle\} \cup \varepsilon, & \text{sonst} \end{cases}$$
$$\text{mit} \quad \upsilon'' = \mathsf{findVar}\,\gamma\,x\,\upsilon$$
$$\upsilon' = \mathsf{findSel}\,\gamma\,(x\,.\,N)\,\upsilon''$$
$$\varepsilon = \{\langle \upsilon, \upsilon'''\rangle \mid \mathsf{isPred}\,\langle \upsilon''', \upsilon'\rangle \wedge \neg \mathsf{isPred}\,\langle \upsilon''', \upsilon\rangle\}$$

Das Prädikat $\mathsf{isPred}(\upsilon_1, \upsilon_2)$ ist wahr, wenn υ_1 ein Vorfahre von υ_2 ist. Die Funktion findVar sucht die Variable im Baum. Da Definitionen von anderen verschattet werden können, wird der Baum vom Blatt ausgehend nach oben durchsucht. Die erste gefundene Definition mit dem Namen der Variablen ist somit die gesuchte Definition. Da nur die gruppengebundenen verwendeten Namen betrachtet werden, muss die Variable im Baum vorhanden sein.

Die Funktion parent gibt den Elternknoten des Knotens υ im Baum γ zurück. top gibt die Wurzel des Baums zurück und chNames gibt die Selektornamen aller Kinder zurück.

$$\mathsf{findVar} :\ \gamma \to x \to \upsilon \to \upsilon$$
$$\mathsf{findVar}\ \gamma\ x\ \upsilon = \begin{cases} \upsilon', & \text{falls } x \in \mathsf{chNames}\ \upsilon' \\ \mathsf{findVar}\ \gamma\ x\ \upsilon', & \text{sonst} \end{cases}$$
$$\text{mit} \quad \upsilon' = \mathsf{parent}\ \gamma\ \upsilon$$

Da Definitionen von anderen verschattet werden können, wird der Baum vom Blatt ausgehend nach oben durchsucht. Die erste gefundene Definition mit dem Namen der Variablen ist somit die gesuchte Definition. Da nur die gruppengebundenen verwendeten Namen betrachtet werden, muss die Variable im Baum vorhanden sein. Die Funktion parent gibt den Elternknoten des Knotens υ im Baum γ zurück und chNames gibt die Selektornamen aller Kinder zurück.

Im zweiten Schritt sucht die Funktion findSel die Selektoren, bis entweder die gesamte Selektorkette gefunden wurde, oder der Knoten keine Kinder hat. Der letzte Knoten ist der Zielknoten der Abhängigkeitskante. Wenn ein Knoten eine Gruppe repräsentiert, diese aber nicht den gesuchten Selektor enthält, so ist der Selektor nicht definiert und es handelt sich um einen Fehler. Die Funktion child gibt das Kind zurück, das den angegebenen Selektor repräsentiert.

$$\mathsf{findSel}: \ \gamma \to N \to \upsilon \to \upsilon$$

$$\mathsf{findSel}\,\gamma\,x.N\,\upsilon = \begin{cases} \upsilon, & \text{falls } \upsilon \in \mathsf{leaves}(\gamma) \\ \mathsf{findSel}\,\gamma\,N\,\upsilon', & \text{falls } \upsilon' = \mathsf{child}\ \gamma\ \upsilon\ x \\ \mathit{Error}, & \text{sonst} \end{cases}$$

Der Graph ergibt sich aus dem Baum und allen für diesen Baum berechneten Abhängigkeitskanten:

$$\mathsf{depGraph}\,\langle \Upsilon, \varepsilon, \upsilon\rangle = \langle \Upsilon, \varepsilon \cup \mathsf{allDepEdges}\,\langle \Upsilon, \varepsilon, \upsilon\rangle, \upsilon\rangle$$

Starke Zusammenhangskomponenten Für einen Abhängigkeitsgraphen werden die starken Zusammenhangskomponenten vom Algorithmus von Sharir [3] berechnet. Dieser liefert eine Liste von Bäumen. Jeder dieser Bäume stellt eine Zusammenhangskomponente dar. Da nur die in diesen Bäumen enthalten Knoten benötigt werden, wird nur die Menge der Knoten jedes Baumes betrachtet. Somit definieren wir die Funktion, die die Zusammenhangskomponenten eines Graphen berechnet wie folgt:

$$\mathsf{calcScc}: \ \gamma \to \mathit{scc}$$

Diese Liste enthält alle Definitionen in Auswertungsreihenfolge. Verschrenkt-rekursive Gruppen sind in einer gemeinsamen Menge und müssen somit gemeinsam ausgewertet werden. Mit Hilfe dieser Liste kann die Gruppe G in einen Ausdruck C transformiert werden.

Transformation einer Gruppe Mittels der in der Abhängigkeitsanalyse berechneten Auswertungsreihenfolge wird im letzten Schritt mittels der Funktion $\mathcal{T}$ die Gruppe G in einen Ausdruck C umgewandelt.

$$\mathcal{T}: \ E \to N \to \mathit{scc} \to \gamma \to C$$

$$\mathcal{T}[\![\{\, x_i = E_i^{\ell_i}{}^{i\in 1..m} \,\}]\!]N\ \mathit{scc}\,\gamma = \mathsf{Seq}\,N\,(\mathsf{calcCdren}\ \ell_i^{\ i\in 1..m}\ \mathit{scc}\ \gamma)$$

$$\mathcal{T}[\![T^{\ell}]\!]N.x\ \mathit{scc}\,\gamma = \begin{cases} \mathtt{Def}\,\langle (N.x), \mathcal{S}^T[\![T]\!]\rangle, & \text{falls } x \notin \mathit{FN}(\ell) \\ \mathtt{RecDef}\,\{\langle N.x, \mathcal{S}^T[\![T]\!]\rangle\}, & \text{sonst} \end{cases}$$

Die Funktion $\mathcal{T}$ baut für nicht rekursive Zusammenhangskomponenten einen Teilbaum auf. Alle Gruppen sind `Seq`, wobei sich die Kinder aus den Zusammenhangskomponenten der Definitionen ergeben. Dies wird von der Funktion calcCdren übernommen. Handelt es sich nicht um eine Gruppe, so ist es eine Definition bzw. eine rekursive Definition. Die Funktion calcCdren ermittelt die Reihenfolge der Kinder eines Knotens und erstellt die entsprechenden Knoten für die Kinder.

$$
\begin{aligned}
&\mathsf{calcCdren}: \ \mathbb{P}\,\ell \to scc \to \gamma \to C^* \\
&\mathsf{calcCdren}\ \emptyset\ scc\ \gamma \qquad\quad = \Diamond \\
&\mathsf{calcCdren}\ \{\ell_i^{i\in 1..n}\}\ scc\ \gamma = C :: (\mathsf{calcCdren}\ \{\ell_i \mid \langle N_i, \ell_i\rangle \notin \Upsilon\}\ scc\ \gamma) \\
&\qquad\qquad \text{mit} \quad \Upsilon = \mathsf{firstScc}\ \ell_i^{i\in 1..n}\ scc \\
&\qquad\qquad\qquad C = \begin{cases} \mathcal{T}[\![EXP(\ell)]\!]\ N\ scc, & \text{falls } \Upsilon = \{\langle N, l\rangle\} \\ \mathsf{transfRec}\ \Upsilon\ scc\ \gamma, & \text{sonst} \end{cases}
\end{aligned}
$$

Für die Reihenfolge wird die erste Zusammenhangskomponente der Kinder von der Funktion firstScc ermittelt. Diese Menge enthält alle Kinder der aktuellen Gruppe, die als nächstes berechnet werden müssen. Ist die Menge einelementig, so handelt es sich um keine rekursive Komponente und somit wird der Teilbaum des Kindes mit der Funktion $\mathcal{T}$ erstellt. Ist es eine rekursive Zusammenhangskomponete, so sind mehrere Kinder in dieser enthalten. Alle Knoten, die in der Zusammenhangskomponente enthalten sind, werden mittels der Funktion transfRec gemeinsam transformiert. Die restlichen Kinder werden im nächsten rekursiven Aufruf behandelt. Sind keine weiteren Kinder vorhanden, so ist das Ende der Liste der Kinder erreicht. Da alle Kinder, die in der Menge Υ enthalten sind in diesem Schritt transformiert werden, müssen sie in den folgenden rekursiven Aufrufen nicht mehr betrachtet werden und ihre Indizes werden aus der Indexmenge gelöscht.

Die Funktion transfRec transformiert die rekursiven Definitionen. Für eine rekursive Zusammenhangskomponente gibt es zwei Möglichkeiten:

1. verschränkt rekursive Funktionen innerhalb einer Gruppe
2. verschränkt rekursive Gruppen

$$
\begin{aligned}
&\mathsf{transfRec}: \ \Upsilon \to scc \to \gamma \to C \\
&\mathsf{transfRec}\ \{\langle N_i, \ell_i\rangle^{i\in 1..n}\}\ scc\ \gamma = \begin{cases} \mathsf{Seq}\ N_{grps}\ C, & \text{falls } \Upsilon_{grps} \neq \emptyset \\ (\mathsf{recC}\ \Upsilon_{recT}\ scc), & \text{sonst} \end{cases} \\
&\quad \text{mit} \qquad C = C_{notRec} +\!\!+ C_{rec} \\
&\qquad\qquad \Upsilon_{grps} = \{\langle N_i, \ell_i\rangle \mid EXP(\ell_i) \in G\} \\
&\qquad\qquad N_{grps} = N_1, \ldots, N_k \quad \text{mit} \quad \langle N_j, \ell_j\rangle \in \Upsilon_{grps} \\
&\qquad\qquad \Upsilon_{recT} = \{\langle N_i, \ell_i\rangle \mid EXP(\ell_i) \in T\} \\
&\qquad\qquad L = \{\ell \mid (y = E^{\ell}) \in EXP(\ell') \wedge \langle N', \ell'\rangle \in \Upsilon_{grps}\} \setminus \{\ell_i^{\,i\in 1..n}\} \\
&\qquad\qquad C_{notRec} = \mathsf{calcCdren}\ L\ scc\ \gamma \\
&\qquad\qquad \gamma' = \mathsf{subGraph}\ \gamma\ \Upsilon_{recT} \\
&\qquad\qquad scc' = \mathsf{calcScc}\ \gamma' \\
&\qquad\qquad C_{rec} = \mathsf{map}\ \mathsf{recC}\ scc'
\end{aligned}
$$

Im ersten Fall, sind keine Gruppen in der Zusammenhangskomponente enthalten und es handelt sich um eine rekursive Definition. Die Transformation aller rechten Seiten und das Erzeugen der rekursiven Definition wird von der Funktion recC übernommen.

Im zweiten Fall handelt es sich um eine rekursive Definition. Alle in der Zusammenhangskomponente enthaltenen Gruppen werden in diesem Knoten zusammengefasst und alle Kinder dieser Gruppen werden an diesen Knoten ange-

hängt. Es müssen nicht alle Definitionen dieser Gruppen in der Zusammenhangskomponente enthalten sein. Dies können alle Definitionen sein, die von keiner der rekursiven Definitionen der Zusammenhangskomponente abhängen. Diese können in einer früheren Komponente stehen. Da diese aber ebenfalls Kinder des rekursiven Knotens werden müssen, müssen sie auch in diesem Schritt transformiert und eingehängt werden. Dazu wird die Menge L, die alle diese Definitionen enthält, berechnet und alle in ihr enthaltenen Knoten werden transfomiert und ergeben C_{notRec}.

Außerdem müssen alle in der Zusammenhangskomponente enthaltenen Definitionen transformiert werden. Durch die zusätzlichen Kanten zu den Vorfahren sind neben den echten rekursiven Funktionen auch Definitionen enthalten, die diese lediglich benutzen. Um auch diese Reihenfolge korrekt zu behandeln, wird die Funktion $\mathsf{calcScc}$ auf dem Teilgraph γ' des Abhängigkeitsgraphen, der nur die Knoten Υ_{recT} enthält, aufgerufen. Diese Komponenten werden von der Funktion recC transformiert.

Die Funktion recC transformiert Zusammenhangskomponenten und entscheidet dabei, ob es sich um eine rekursive Zusammenhangskomponente handelt. Alle mehrelementigen Komponenten sind rekursive Zusammenhangskomponenten und werden in eine gemeinsame rekursive Definition transformiert.

Alle übrigen Komponenten können entweder rekursive Funktionen sein, oder normale Definitionen. Daher kann für diese die Funktion $\mathcal{T}$ genutzt werden.

$$
\begin{aligned}
&\mathsf{recC} : \ \Upsilon \to scc \to C \\
&\mathsf{recC}\ \{\langle N, \ell\rangle\}\ scc \quad\ = \mathcal{T}[\![EXP(\ell)]\!]\ N\ scc \\
&\mathsf{recC}\ \{\langle N_i, \ell_i\rangle^{i\in 1..n}\}\ scc = \texttt{RecDef}\ (N_i\ \mathcal{S}^T[\![EXP(\ell_i)]\!])^{i\in 1..n}
\end{aligned}
$$

4 Denotationelle Semantik

Dem Resultat der Abhängigkeitsanalyse aus dem vorherigen Kapitel ordnen wir nun mittels einer Auswertungsfunktion $\mathcal{E}$ eine Interpretation zu.

Wir nutzen die etablierten Techniken der denotationellen Semantik [4,5,6] ohne zu sehr ins Detail zu gehen.

4.1 Semantische Bereiche und Hilfsfunktionen

Wir ordnen jedem Ausdruck A einen Wert aus den folgenden semantischen Bereichen zu:

$$
\begin{aligned}
\mathsf{V} &= \mathsf{N} + \mathsf{B} + \mathsf{S} + \Gamma + \mathsf{F} \\
\Gamma &= x \hookrightarrow \mathsf{V} \\
\mathsf{F} &= \mathsf{V} \to \mathsf{V}
\end{aligned}
$$

Die Menge aller Werte V besteht aus Zahlen N, Wahrheitswerten B, Zeichenketten S sowie Gruppenwerten Γ und Funktionswerten F.

Gruppen(werte) sind partielle Funktionen mit endlichem Definitonsbereich, die Variablen auf Werte abbilden. Intuitiv bildet eine Gruppe die linke Seite einer Definiton auf den Wert der rechten Seite ab. Die Menge der Gruppen ist

auch gleichzeitig die Menge der Auswertungskontexte, die freie Variablen eines Ausdrucks an Werte bindet.

Funktionswerte sind Funktionen die Werte in Werte abbilden.

Weiterhin definieren wir die drei Hilfsfunktionen $\lhd$, build und update.

- Der Operator $\lhd$ vereinigt zwei partielle Funktionen Γ_1 und Γ_2, wobei die rechte Abbildung ggf. Zuordnungen der linken überschreibt:

$$\lhd : \Gamma \to \Gamma \to \Gamma$$
$$(\Gamma_1 \lhd \Gamma_2)(x) = \begin{cases} V, & \text{falls } \Gamma_2(x) \text{ definiert und } \Gamma_2(x) = V \\ \Gamma_1(x), & \text{sonst} \end{cases}$$

- Die Funktion build erweitert einen gegebenen Kontext um Bindungen für alle Variablen, die ein Ausdruck, der rechts des Namens $x_1 . \cdots . x_m$ definiert ist, zur Verfügung hat.

$$\mathsf{build} : \Gamma \to N \to \Gamma$$
$$\mathsf{build}\ \Gamma\ (x_1 . \cdots . x_m) = \Gamma \lhd \left(\lhd_{i \in 1..m} \Gamma(x_1) \cdots (x_i) \right)$$

- Die Funktion update trägt einen Wert unter dem Namen $x_1 . \cdots . x_m . x$ in die Gruppe Γ ein. Dabei wird pro Selektor eine Gruppe erweitert, die jeweils in die umschließende Gruppe eingetragen werden muss.

$$\mathsf{update} : \Gamma \to N \to x \to V \to \Gamma$$
$$\mathsf{update}\ \Gamma\ (x_1 . \cdots . x_m)\ x\ V = \Gamma_0$$
$$\text{mit} \quad \Gamma_m = \Gamma(x_1) \cdots (x_m) \lhd \{x \mapsto V\}$$
$$\Gamma_i = \Gamma(x_1) \cdots (x_i) \lhd \{x_{i+1} \mapsto \Gamma_{i+1}\} \quad \text{für } i \in 0..m-1$$

4.2 Auswertungsfunktion

Wir zeigen die Auswertungsfunktion $\mathcal{E}$ in zwei Teilen; Abb. 19 zeigt den interessanteren Teil, der Gruppen auswertet, Abb. 20 zeigt der Vollständigkeit halber die Definitionen für die üblichen Ausdrücke Abstraktion, Applikation, Fallunterscheidung usw.

Bevor wir die einzelnen Fälle im Detail erläutern, halten wir eine wichtige Gemeinsamkeit fest: Gruppen werden zu partiellen Funktionen ausgewertet. Diese partiellen Funktionen werden auch genutzt, um den aktuellen Auswertungskontext zu fassen. Allen drei Konstruktionen, die zu Gruppen auswerten, ist gemeinsam, dass sie solch eine partielle Funktion liefern, die eine Erweiterung des aktuellen Auswertungskontexts ist. Da Gruppen Schritt für Schritt aufgebaut werden, wird das zweite Argument von $\mathcal{E}$ auch dazu genutzt, Informationen von einem Auswertungsschritt zum nächsten zu geben.

Wir erläutern nun die einzelnen Fälle im Detail:

- `Def` $N.x\ S$: Bei `Def` $N.x\ S$ muss der Ausdruck S ausgewertet werden und unter dem Name $N.x$ in den aktuellen Kontext eingefügt werden. Dazu wird mittels build der geltende Auswertungskontext für S aufgebaut und das Ergebnis mittels update in das Result eingetragen.

$$
\begin{array}{l}
\mathcal{E} : \; A \to \Gamma \to \mathsf{V} \\
\hline
\mathcal{E}[\![\texttt{Def}\ N\texttt{.}x\ S]\!]\Gamma \qquad\qquad\qquad\qquad = \mathsf{update}\ \Gamma\ N\ x\ \ (\mathcal{E}[\![S]\!](\mathsf{build}\ \Gamma\ N)) \\
\\
\mathcal{E}[\![\texttt{Seq}\ N_i\texttt{.}x_i{}^{i\in 1..m}\ C_j{}^{j\in 1..p}]\!]\Gamma = \Gamma_p \\
\qquad\qquad\qquad\qquad \text{mit} \quad \Gamma'_0 = \Gamma \\
\qquad\qquad\qquad\qquad\qquad\quad \Gamma'_i = \mathsf{update}\ \Gamma'_{i-1}\ N_i\ x_i\ \varnothing \quad \text{für } i \in 1..m \\
\qquad\qquad\qquad\qquad\qquad\quad \Gamma_0 = \Gamma'_m \\
\qquad\qquad\qquad\qquad\qquad\quad \Gamma_j = \mathcal{E}[\![C_j]\!]\Gamma_{j-1} \quad \text{für } j \in 1..p \\
\\
\mathcal{E}[\![\texttt{RecDef}\ (N_i\texttt{.}x_i\ S_i)^{i\in 1..m}]\!]\Gamma = \Gamma_m \\
\qquad\qquad\qquad\quad \text{mit} \quad \Phi\left\langle \mathsf{F}_i{}^{i\in 1..m} \right\rangle = \left\langle \mathcal{E}[\![S_i]\!](\mathsf{build}\ N_i\ \Gamma'_m)^{i\in 1..m} \right\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{mit} \quad \Gamma'_0 = \Gamma \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \Gamma'_i = \mathsf{update}\ \Gamma'_{i-1}\ N_i\ x_i\ \mathsf{F}_i \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{für } i \in 1..m \\
\qquad\qquad\qquad\qquad\quad \left\langle \mathsf{F}_i{}^{i\in 1..m} \right\rangle = \mathsf{FIX}\Phi \\
\qquad\qquad\qquad\qquad\qquad\quad \Gamma_0 = \Gamma \\
\qquad\qquad\qquad\qquad\qquad\quad \Gamma_i = \mathsf{update}\ \Gamma_{i-1}\ N_i\ x_i\ \mathsf{F}_i \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{für } i \in 1..m
\end{array}
$$

Abbildung 19. Auswertungsfunktion für Gruppen.

- $\texttt{Seq}\ N_i\texttt{.}x_i{}^{i\in 1..m}\ C_j{}^{j\in 1..p}$: Beim Auswerten einer Sequenz werden neue Gruppen unter den Namen $N_i\texttt{.}x_i{}^{i\in 1..m}$ in den Kontext eingefügt. Diese werden zunächst leer mittels update im Kontext Γ'_m angelegt. In diesem Kontext werden dann der Reihe nach alle $C_j{}^{j\in 1..p}$ ausgewertet, wobei jeweils die Auswertung von C_{j-1} den Kontext für die Auswertung von C_j liefert. Diese Verkettung ist nötig, da die $C_j{}^{j\in 1..p}$ in Abhängigkeitsreihenfolge stehen und weiter rechts stehende C_j Definitionen aus links von ihnen stehenden C_j nutzen können.
- $\texttt{RecDef}\ (N_i\texttt{.}x_i\ S_i)^{i\in 1..m}$: Der Fall (gegenseitig) rekursiver Definitionen ist am komplexesten. Wie in der denotationellen Semantik üblich, weisen wir rekursiven Funktionen den kleinsten Fixpunkt eines Funktionals Φ zu. In unserem Falle besteht dieses Funktional aus der Auswertung der rechten Seiten der Funktionen in einem Kontext, der durch zwei Schritte aufgebaut wird: Erst werden die Argumente des Funktionals, d. h. die rekursiven Funktionen, unter den Namen $N_i\texttt{.}x_i{}^{i\in 1..m}$ mittels update eingetragen. Dies schnürt den „rekursiven Knoten“. Dann wird für jede rechte Seite S_i die soeben aufgebaute Umgebung um die Menge aller verfügbaren Variablen entlang des Namens N_i erweitert.

  Das Resultat der Auswertung eines RecDef ist eine Gruppe, in der unter den Namen $N_i\texttt{.}x_i$ die Komponenten F_i des Fixpunkts, also die rekursiven Funktionen, eingetragen sind.

$$
\begin{array}{ll}
\mathcal{E}[\![x]\!]\Gamma & = \Gamma(x) \\
\mathcal{E}[\![A\,.\,x]\!]\Gamma & = (\mathcal{E}[\![A]\!]\Gamma)(x) \\
\mathcal{E}[\![A_1\ A_2]\!]\Gamma & = (\mathcal{E}[\![A_1]\!]\Gamma)(\mathcal{E}[\![A_2]\!]\Gamma) \\
\mathcal{E}[\![\lambda x\,.\,A]\!]\Gamma & = \mathsf{F} \qquad \text{mit} \quad \mathsf{F}(\mathsf{V}) = \mathcal{E}[\![A]\!](\Gamma \lhd \{x \mapsto \mathsf{V}\}) \\
\mathcal{E}[\![\texttt{IF}\ A_1\ \texttt{THEN}\ A_2\ \texttt{ELSE}\ A_3]\!]\Gamma & = \begin{cases} \mathcal{E}[\![A_2]\!]\Gamma, & \text{falls } \mathcal{E}[\![A_1]\!]\Gamma = \texttt{true} \\ \mathcal{E}[\![A_1]\!]\Gamma, & \text{falls } \mathcal{E}[\![A_1]\!]\Gamma = \texttt{false} \end{cases} \\
\mathcal{E}[\![\alpha]\!]\Gamma & = \alpha \qquad \text{für } \alpha \in \{n, b, s\}
\end{array}
$$

Abbildung 20. Auswertungsfunktion für einfache Ausdrücke.

5 Verwandte Arbeiten

Es existieren unterschiedliche Ansätze für eine flexible Modularisierung in statisch wie dynamisch getypten funktionalen Sprachen.

Für Haskell wird in [8] ein Modulsystem vorgestellt, welches Records und Module gleichsetzt. Diese Recordmodule sind, genau wie die Gruppen in unserem Ansatz, First-class-values mit einer Dot-Notation. Da Haskell typisiert ist, sind auch diese Records typisiert. Dieser Typ muss für jeden Record angegeben werden und gibt unter anderem Auskunft über die in diesem Modul definierten Namen. Durch Typinferenz kann für jeden Ausdruck ermittelt werden, welche Selektionen erlaubt sind. Da es sich bei Haskell um eine Lazy funktionale Sprache handelt, ist eine Abhängigkeitsanalyse nicht nötig.

In SML gibt es eine strikte Trennung zwischen Modul- und Berechnungssprache. Die Flexibiltät der ML-Module beruht auf Modulfunktoren, die es ermöglichen Module zu erzeugen und zu verändern. Das SML Modulsystem hat keine Möglichkeit verschränkt rekursive Module zu definieren. Es gibt einige Erweiterungen für dieses Modulsystems, die verschränkt rekursive Module erlauben.

Der weitreichenste Ansatz für rekursive Module in ML sind Mixin Modules [7]. Allerdings müssen innerhalb eines Moduls alle benötigten äußeren Definitionen deklariert werden. Mixin Modules sind wie alle ML Module keine First-class-values. Die Abhängigkeiten zu anderen Modulen wird in der Signatur des Moduls angegeben. Mittels der Funktion „sum“ können Module, die zusammenpassen, verklebt werden. Dazu müssen die Module gegenseitig die benötigten Funktionen zur Verfügung stellen. Die Zuordnung der fehlenden Definitionen erfolgt somit beim Verkleben. Damit unterscheidet sich dieser Ansatz grundlegend von den Gruppen, da alle Abhängigkeiten explizit angegeben werden müssen.

In Ocaml [9] sind die Module, im Gegensatz zu SML, First-class-values. Allerdings sind auch hier, anderes als bei GLang, keine rekursiven Module erlaubt.

Flatt und Felleisen stellen in [12] eine Sprache für units vor. Diese ermöglicht es statisch und dynamisch typisiert Module für ML- oder Scheme-ähnliche Sprachen zu definieren. Auch hier handelt es sich, wie in ML, um eine eigenständige Sprache für Module, womit diese keine First-class-values sind. Innerhalb einer

Unit werden alle Information für eine getrennte Compilierung und das Linking zu anderen Modulen gespeichert. Somit müssen auch hier die Abhängigkeiten nach außen explizit angegeben werden. Für die Units sind verschränkte Abhängigkeiten über die Modulgrenzen hinweg erlaubt.

In den meisten Sprachen muss die Auswertungsreihenfolge vom Programmierer explizit angegeben werden. Eine Ausnahme stellt die Sprache Modula-3 [11] dar. In Modula-3 wird, wie für GLang, die Auswertungsreihenfolge mittels eines Abhängigkeitsgraphen ermittelt, allerdings sind rekursive Module in dieser Sprache nicht erlaubt.

Claus Reinke hat in seiner Dissertation [10] ein Modulsystem für funktionale Call-by-value-Sprachen entwickelt. Die in der Arbeit vorgestellten Frames entsprechen weitgehend unseren Gruppen. Insbesondere sind es dynamisch typisierte First-class-values. Frames können eine Menge von verschränkt rekursiven Definitionen enthalten. Eine verschränkte Rekursion über Framegrenzen hinweg ist jedoch nicht möglich. Namen anderer Module müssen grundsätzlich importiert werden. Durch diese Imports wird die Auswertungsreihenfolge vom Programmierer vorgegeben. Damit sind verschränkt rekursive Module nicht direkt möglich. Sie können durch Funktionen nachgebildet werden, indem die benötigten Module als Funktionsparameter übergeben werden.

6 Fazit und Ausblick

Das vorgestellte Konzept der Gruppen stellt einen sehr flexiblen, ausdruckstarken und einheitlichen Mechanismus für dynamische Datenstrukturen, Bindungen und Modularisierung zur Verfügung. Um diese Ausdrucksmächtigkeit effizient ausführen zu können, wurde eine Abhängigkeitsanalyse entwickelt, die eine Call-by-value-Auswertung der Gruppen ermöglicht. Für die sich aus der Analyse ergebende Zwischenrepräsentation mit expliziter Auswertungsreihenfolge wurde eine denotationelle Semantik definiert.

Wir haben die in diesem Beitrag vorgestellte Sprache GLang bereits um Gruppenmorphismen erweitert. Diese ermöglichen es, nicht nur Gruppen zur Laufzeit zu erzeugen, sondern auch bestehende Gruppen zu erweitern oder einzuschränken. Dies verbessert die Möglichkeiten der Wiederverwendbarkeit und die Flexibilität von Gruppen. Desweiteren haben wir einen auf Kontrollflussanalyse basierenden Mechanismus für Importe sowie Möglichkeiten zur Einschränkung von Sichtbarkeiten beim Importieren und Exportieren entwickelt und implementiert.

Zur Zeit arbeiten wir an an einer Übersetztung der Zwischenrepräsentation mit expliziter Reihenfolge in eine ML-ähnliche Sprache mit „let“ und „letrec“ sowie Records und entwickeln ein Konzept zur separaten Übersetzung.

Dank Wir bedanken uns bei Peter Pepper und Christoph Höger für wertvolle Diskussionen und Hinweise zum Thema Sprachentwurf und Implementierbarkeit sowie bei Doug Smith vom Kestrel Institute, Palo Alto, wo ein Grossteil der vorliegenden Arbeit entstanden ist.

Literatur

1. Pepper, P., Hofstedt, P.: Funktionale Programmierung – Sprachdesign und Programmiertechnik. Springer (2006)
2. Abelson, H., Sussman, G.J., Sussman, J.: Structure and Interpretation of Computer Programs. 2nd edn. MIT Press (1996)
3. Sharir, M.: A strong-connectivity algorithm and its applications in data flow analysis. Computers & Mathematics with Applications **7**(1) (1981) 67 – 72
4. Tennent, R.D.: The denotational semantics of programming languages. Commun. ACM **19**(8) (1976) 437–453
5. Gordon, M.J.C.: The Denotational Description of Programming Languages. Springer-Verlag (1979)
6. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. Computer Science Series. MIT Press (1981)
7. Hirschowitz, T., Leroy, X.: Mixin modules in a call-by-value setting. ACM Trans. Program. Lang. Syst. **27** (September 2005) 857–881
8. Peyton Jones, S.L., Shields, M.B.: First class modules for Haskell. In: 9th International Conference on Foundations of Object-Oriented Languages (FOOL 9), Portland, Oregon. (January 2002) 28–40
9. Frisch, A., Garrigue, J.: First-class modules and composable signatures in objective caml 3.12 (2010)
10. Reinke, C.: Functions, Frames, and Interactions – completing a lambda-calculus-based purely functional language with respect to programming-in-the-large and interactions with runtime environments. PhD thesis, Universität Kiel (1997)
11. Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., Nelson, G.: Modula-3 report (revised). ACM SIGPLAN Notices **27**(8) (1992) 15–42
12. Flatt, M., Felleisen, M.: Units: cool modules for hot languages. In: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation. PLDI '98, New York, NY, USA, ACM (1998) 236–248